TMA4215 Numerical Mathematics - Fall 2020

Project 2

alexaoh, jimoskar

October 2020

1 Introduction

In the first part of this project report, some model choices will be explained and justified.

Then, in the second part, a systematic investigation of optimal choices of the parameters in the neural network will be documented. Data from all four of the supplied test functions on their respective domains will be shown. Based on observations from this data a discussion about which combination of parameters is optimal will be written. Some remarks about the methods used will also be made. Finally, a conclusion about the optimal parameters in the network will be made.

In the third part of the report, the numerical methods symplectic Euler and Størmer-Verlet will be used to calculate the trajectories and the Hamiltonian function of three suggested examples of separable Hamiltonian problems in mechanics. A necessity in this case is to calculate the gradient numerically. The algorithm used for this is presented also.

In the fourth part, the model will be used to approximate the unknown trajectories and Hamiltonian function of the supplied data.

Finally, a short discussion about the code architecture will be made, to justify some of our choices when implementing the necessary code in Python.

2 Model Choices

2.1 Embedding of Input

When the dimension, d, of the layers in the network is greater than the dimension of the input data, d_0, we need to embed the input in d dimensions. There are probably many ways to do this, e.g. if the input is given by $p = [x, y]^T$ and the dimension of the network is $d = 4$, one could perhaps embed p as $[x, y, x, y]^T$, $[x, x, y, y]^T$ or $[x, y, 0, 0]^T$ to mention some possibilities. We experimented with the second option, with good results, but we ended up using the last embedding, i.e. appending zeros in the remaining coordinates, as this was the easiest to implement and generalize over all different combinations of d and d_0, such that the trained model becomes a well defined function mapping input to output.

2.2 Stochastic Gradient Descent

Throughout the project, Stochastic Gradient Descent (henceforth SGD) has been used. This means that instead of running all the training input through the network at once, we sift through the data by randomly drawing, without replacement, a chunk of input points $\bar{I} < I$ at each iteration of the training algorithm. This makes the training process more time efficient, as seen from data presented later.

2.3 Optimization Method

The ADAM method has been used as the optimization tool throughout the tests presented, as opposed to the standard gradient descent method (henceforth Vanilla method). The latter is given by

$$\theta^{(r+1)} = \theta^{(r)} - \tau \nabla_\theta J(\theta^{(r)}), \tag{1}$$

where θ are the parameters that should be optimized, $\theta^{(0)}$ is the initial guess and τ is the "learning parameter", which acts as a step size in the direction of the gradient of J with respect to θ.

From various tests we have seen that the Vanilla method gives faster convergence than ADAM in some instances (e.g. with scaling in figure 2), but it seemed to be much less stable, meaning that it has random spikes in the convergence of the objective function. In figure 1, the mean of the objective function over 15 different training sessions (with different training data) is plotted against the iterations in the training process. One clearly sees the unstable behavior of the Vanilla method, with $\tau = 0.01$. When τ is increased, the objective function typically diverges, and when τ is further decreased from 0.01, it typically converges slower than the ADAM method (based on our experience). Consequently, we have consistently used the ADAM method throughout the testing for optimal parameters.

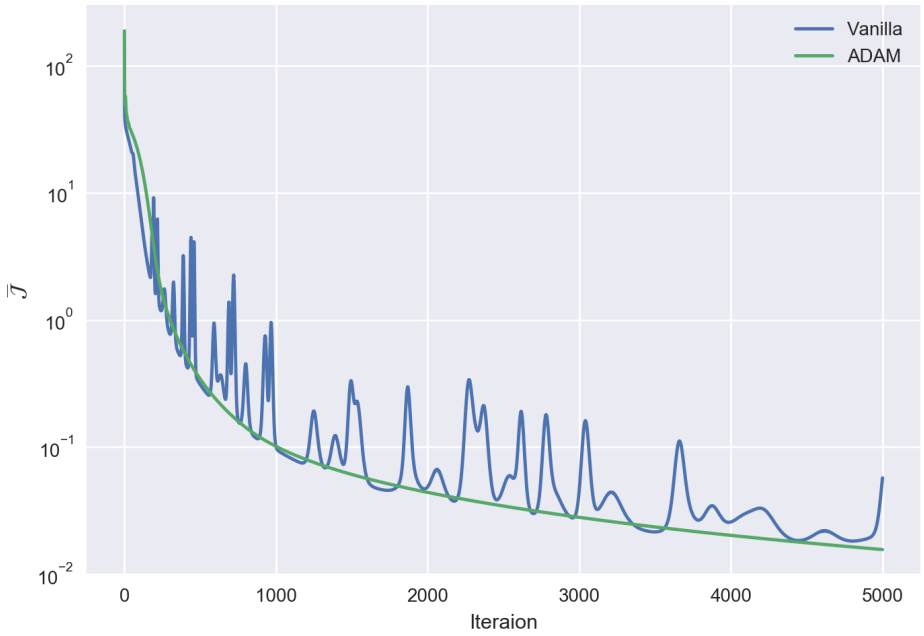


Figure 1: The mean of the objective function over 15 training sessions plotted against the iterations for both the Vanilla gradient method and ADAM descent method. Here, $\tau = 0.01$, $I = 1000$, $K = 20$, $h = 0.05$ and $d = 4$. We used test function 3 and SGD with a chunk of $\bar{I} = 100$.

2.4 Scaling

The primary reason for scaling the input data is probably to ensure that the range of the hypothesis function $\eta(x)$ contains the range of the output (i.e. the c_i's). When $\eta(x) = x$, range$(\eta) = \mathbb{R}$ if we assume that the input is real. Hence, the hypothesis function's range must contain the range of the output. Then the primary reason for scaling is no longer pressing, and although it is generally advised to scale the input data, we had no observable issues related to scaling when omitting it and using $\eta(x) = x$ (except possibly in section 4.3). Furthermore, some simple tests show that the convergence is typically slower when we use scaling and also when we use $\eta(x) = \frac{1}{2}(1 - \tanh(x/2))$, which requires the use of scaling. In figure 2 the average evolution of the objective function is plotted over 5 training sessions. What we see is a typical result: Although the convergence in the sessions with scaling "catch up" to the one without, it requires more iterations (over 6000 in this case). Including scaling in our search for optimal parameters would hence yield much longer run times, probably with little benefit. We have therefore omitted scaling and used $\eta(x) = x$ in our systematic tests to find optimal parameters.

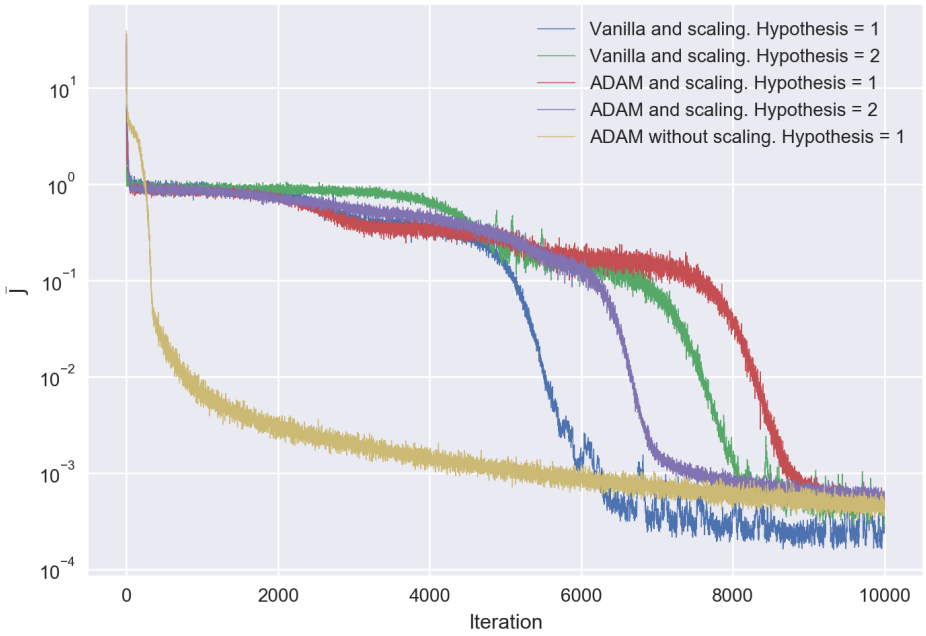


Figure 2: The average evolution of the objective function J over 5 training sessions on the second test function, $F(y) = 1 - \cos y$. "Hypothesis = 1" refers to when $\eta(x) = x$, while "Hypothesis = 2" refers to when $\eta(x) = \frac{1}{2}(1 - \tanh(x/2))$. Here, $\tau = 0.01$, $I = 500$, $K = 23$, $h = 0.1$ and $d = 2$. We used test function 2 and SGD a chunk $= \bar{I} = 50$.

3 Results and Discussion Regarding Suggested Functions

Some results from the experiments with the ResNet neural network are shown in this section. After some initial manual testing on the first suggested test function, a range of parameters for further testing was defined, based on the initial observations.

All combinations of parameters shown in table 1 were tested in the following way: New neural networks were instantiated for each combination of parameters. Then, the neural network was trained and the value of the objective function in the last iteration of the training was saved. This value is regarded as an indication of which combination converges faster than others, in order to favor some sets of parameters over others. For every combination of the parameters networks were trained, then tested with new data. Similar to in the training case, the last value of the objective function, as well as the ratio of the points that were correctly classified, were saved. This ratio was calculated based on a tolerance of 0.005, i.e. if the absolute value of each point approximated from the network and its exact answer is less than the tolerance of 0.005, the point is regarded as correctly classified.

The results from the described investigation of the model on the four suggested functions are displayed in the subsections to come. In the end of this section, a conclusion will be

drawn about which of the tested combinations of parameters seemed to give the best results, based solely on the four suggested test functions on their respective domains.

During the training and testing on the functions, the ADAM method was used for optimization. As noted in section one, the ADAM method was observed as more stable than the Vanilla method, which makes it more suitable for use. Furthermore, the simulations were run with 2000 iterations and with $I = 1000$ points. Both the training and the testing of the neural networks were run with and without SGD, in order to test how it affects the run time and the performance of the classifications. When using SGD, a chunk of $\bar{I} = 100$ points was used in each iteration.

The data is displayed in tables which are sorted in ascending order according to the fourth column, which is the value of the objective function in iteration 2000.

Moreover, one figure is shown per test function. This contains a convergence plot and a comparison between the correct graph and the approximated graph in each case, using the uppermost combinations in tables 5, 9, 13 and 17 respectively. These are qualitative examples of the fact that the graph can be approximated well by the model using SGD, while giving a faster run time compared to when all points are run through at once.

Table 1: All combinations of the given parameter values are tested, on the four suggested test functions.

Quantity	Value(s)
K	10, 15, 20, 23, 30
d	2, 3, 4
h	0.05, 0.1, 0.2, 0.3, 0.4

3.1 First test function: $F(y) = \frac{1}{2}y^2$

The results from the investigation of the first suggested function are shown in the tables 2 - 5.

Figure 3 shows the convergence plot and the comparison between the correct function graph and the approximation produced by one realization of the model with the uppermost parameter configuration in table 5.

Table 2: Top 6 combinations from **training** with data from $F(y) = \frac{1}{2}y^2$, with static parameters as described in the introduction to this section. All the points I were run through the network at once when generating this data.

K	d	h	J
10	3	0.1	0.003271561196348852
30	3	0.1	0.004629531959077177
15	4	0.05	0.0052069189531486785
15	3	0.05	0.005296849937155035
10	2	0.1	0.005367696319488213
20	4	0.05	0.005804047738868283

Table 3: Top 6 combinations from **training** with data from $F(y) = \frac{1}{2}y^2$, with static parameters as described in the introduction to this section. **SGD** was used in the training to generate this data, with chunks of points as described earlier.

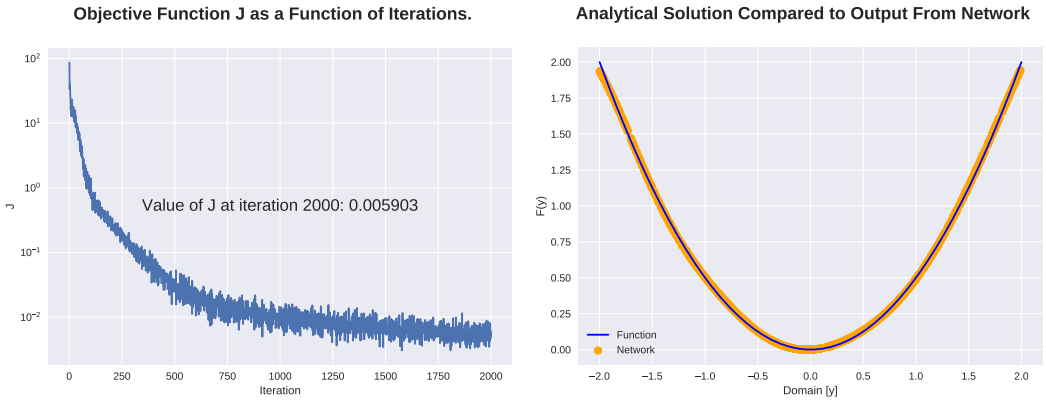
K	d	h	J
20	4	0.3	0.0007853135863417866
10	2	0.3	0.0011480386941942532
15	4	0.3	0.001325579312236484
20	2	0.3	0.0016307517046253409
23	4	0.4	0.0016629452659533586
20	3	0.2	0.0016650094685827293

Table 4: Top 6 combinations from **testing** with data from $F(y) = \frac{1}{2}y^2$, with static parameters as described in the introduction to this section. All the points I were run through the network at once when generating this data.

K	d	h	J	Ratio
30	4	0.1	0.002065263260821445	0.989
20	4	0.1	0.002721769906202256	0.966
10	3	0.2	0.0034841446007803844	0.959
23	4	0.1	0.004336073119820962	0.953
30	3	0.1	0.00477770306838583	0.959
20	4	0.05	0.005362276159961846	0.884

Table 5: Top 6 combinations from **testing** with data from $F(y) = \frac{1}{2}y^2$, with static parameters as described in the introduction to this section. **SGD** was used in the training to generate this data.

K	d	h	J	Ratio
20	4	0.05	0.004855875435160285	0.925
20	2	0.1	0.005866541570337117	0.909
20	2	0.05	0.006790224899888018	0.88
10	4	0.2	0.006919202205239331	0.874
30	3	0.05	0.0071410399652602945	0.891
30	4	0.2	0.0072610789074522685	0.791



(a) Objective function

(b) Comparison

Figure 3: Figures produced by training and testing a neural network with the upper combination of values from table 5. Figure (a) shows the objective function with respect to the iterations and figure (b) shows the exact solution compared to the approximated graph produced by the network. This qualitatively shows that the approximation is fine.

3.2 Second test function: $F(y) = 1 - \cos y$.

The results from the investigation of the second suggested function are shown in the tables 6 - 9.

Figure 4 shows the convergence plot and the comparison between the correct function graph and the approximation produced by one realization of the model with the uppermost parameter configuration in table 9.

Table 6: Top 6 combinations from **training** with data from $F(y) = 1 - \cos y$, with static parameters as described in the introduction to this section. All the points I were run through the network at once when generating this data.

K	d	h	J
20	2	0.1	0.000773886877537679
15	4	0.05	0.004142355153454115
20	3	0.1	0.004222873639860813
10	4	0.1	0.006125037446337861
23	2	0.2	0.00728256945102555
15	3	0.05	0.007356034903953231

Table 7: Top 6 combinations from **training** with data from $F(y) = 1 - \cos y$, with static parameters as described in the introduction to this section. SGD was used to generate this data, with chunks of points as described earlier.

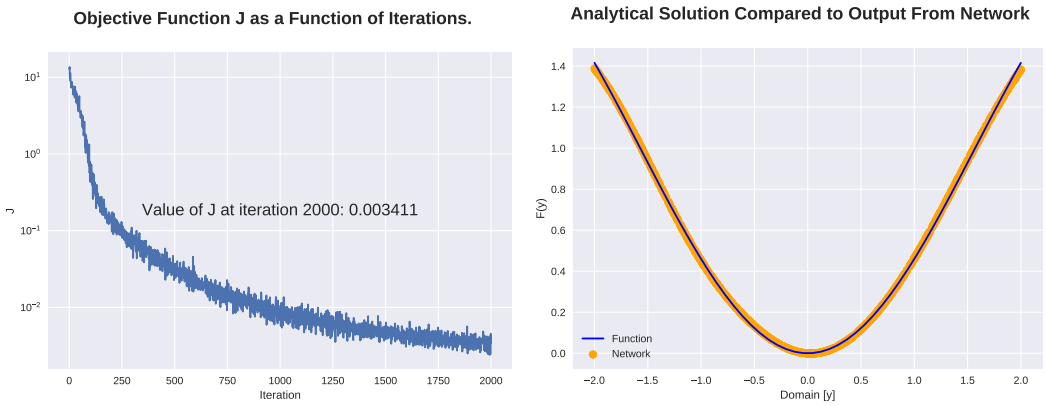
K	d	h	J
23	3	0.3	0.00027240995385181355
15	4	0.4	0.0005096521246452566
30	4	0.3	0.0005117284752766509
20	4	0.2	0.000595171417080602
23	4	0.2	0.0009088073193744871
20	4	0.1	0.0009151724421146553

Table 8: Top 6 combinations from **testing** with data from $F(y) = 1 - \cos y$, with static parameters as described in the introduction to this section. All the points I were run through the network at once when generating this data.

K	d	h	J	Ratio
15	2	0.05	0.0024183397699524193	0.95
10	4	0.2	0.0030987614499304486	0.939
30	3	0.05	0.003645465074784042	0.965
30	4	0.05	0.003735053188659942	0.958
23	2	0.1	0.00486404976506172	0.98
20	4	0.05	0.004955652035532262	0.875

Table 9: Top 6 combinations from **training** with data from $F(y) = 1 - \cos y$, with static parameters as described in the introduction to this section. **SGD** was used in the training to generate this data, with chunks of points as described earlier.

K	d	h	J	Ratio
20	2	0.05	0.0019867854388946216	0.966
30	2	0.1	0.0026077549134037393	0.978
30	3	0.1	0.0027081334181057936	0.957
20	4	0.05	0.003387481783018866	0.943
30	4	0.05	0.0036900089833759725	0.94
30	3	0.05	0.0038131060663503186	0.95



(a) Objective function

(b) Comparison

Figure 4: Figures produced by training and testing a neural network with the upper combination of values from table 9. Figure (a) shows the objective function with respect to the iterations and figure (b) shows the exact solution compared to the approximated graph produced by the network. This qualitatively shows that the approximation is fine.

3.3 Third test function: $F(y) = \frac{1}{2}(y_1^2 + y_2^2)$.

The results from the investigation of the third suggested function are shown in the tables 10 - 13.

Figure 5 shows the convergence plot and the comparison between the correct function graph and the approximation produced by one realization of the model with the uppermost parameter configuration in table 13.

Table 10: Top 6 combinations from **training** with data from $F(y) = \frac{1}{2}(y_1^2 + y_2^2)$, with static parameters as described in the introduction to this section. All the points I were run through the network at once when generating this data.

K	d	h	J
30	3	0.1	0.057960929549445435
23	4	0.1	0.07001944037781303
20	4	0.05	0.07607320471572306
23	4	0.05	0.08108190330391844
23	3	0.05	0.1385186068071021
20	3	0.05	0.1704813592415716

Table 11: Top 6 combinations from **training** with data from $F(y) = \frac{1}{2}(y_1^2 + y_2^2)$, with static parameters as described in the introduction to this section. **SGD** was used to generate this data, with chunks of points as described earlier.

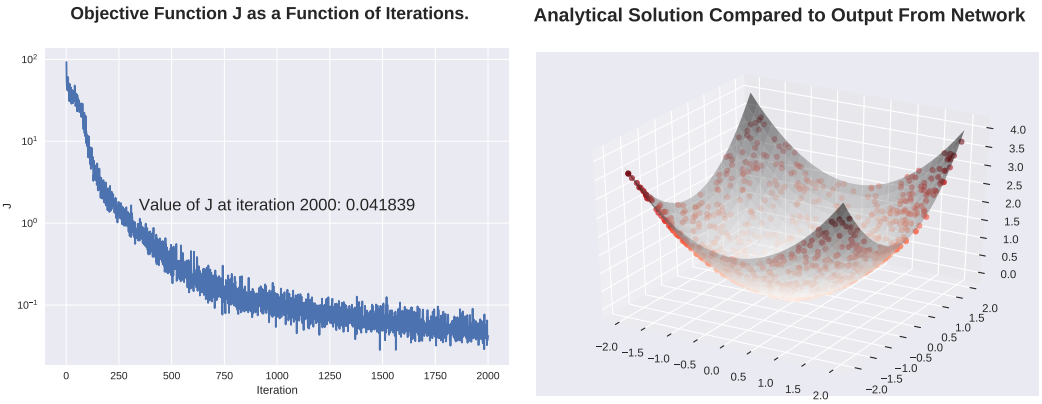
K	d	h	J
20	4	0.2	0.010226810921756606
30	4	0.1	0.013304860997867618
20	3	0.05	0.019251335903161904
10	4	0.3	0.019608874258294003
23	4	0.3	0.02235923607703792
15	4	0.1	0.025260685583598445

Table 12: Top 6 combinations from **testing** with data from $F(y) = \frac{1}{2}(y_1^2 + y_2^2)$, with static parameters as described in the introduction to this section. All the points I were run through the network at once when generating this data.

K	d	h	J	Ratio
30	3	0.1	0.03675764882838929	0.454
10	4	0.05	0.03915548365987047	0.463
15	4	0.1	0.04196298431822519	0.441
20	4	0.2	0.05695400433622645	0.374
30	4	0.1	0.05845823194260694	0.36
30	4	0.2	0.072569376033764	0.301

Table 13: Top 6 combinations from **testing** with data from $F(y) = \frac{1}{2}(y_1^2 + y_2^2)$, with static parameters as described in the introduction to this section. **SGD** was used in the training to generate this data, with chunks of points as described earlier.

K	d	h	J	Ratio
23	4	0.05	0.03536111086256525	0.48
20	4	0.05	0.062005719482834815	0.35
30	4	0.1	0.06765410410688545	0.307
15	3	0.05	0.08758738092045637	0.308
23	4	0.1	0.10640494680713011	0.232
30	3	0.2	0.16696771627804516	0.196



(a) Objective function

(b) Comparison

Figure 5: Figures produced by training and testing a neural network with the upper combination of values from table 13. Figure (a) shows the objective function with respect to the iterations and figure (b) shows the exact solution compared to the scatter plot produced by the network. This qualitatively shows that the approximation is fine.

3.4 Fourth test function: $F(y) = -\frac{1}{\sqrt{y_1^2+y_2^2}}$.

The results from the investigation of the fourth suggested function are shown in the tables 14 - 17.

Figure 6 shows the convergence plot, as well as the comparison between the correct function graph and the approximation produced by one realization of the model with the uppermost parameter configuration in table 17.

The somewhat slow convergence of the objective function for this test function compared to the others indicates that more than 2000 iterations would give significantly better approximations. To show that more iterations give better convergence, figure 7 shows the same plots as figure 6, but with 10000 iterations. We also observe that the unsteadiness of the converge plot increases for the two-dimensional test functions, compared to the one-dimensional cases. This is something that will be considered later in the project also.

Table 14: Top 6 combinations from **training** with data from $F(y) = -\frac{1}{\sqrt{y_1^2+y_2^2}}$, with static parameters as described in the introduction to this section. All the points I were run through the network at once when generating this data.

K	d	h	J
30	4	0.1	0.1581432339546482
23	4	0.2	0.36377024214621445
30	3	0.1	0.7499861512534348
15	4	0.2	1.000097075140732
30	4	0.05	1.1214422169097413
23	4	0.1	1.2080263694526199

Table 15: Top 6 combinations from **training** with data from $F(y) = -\frac{1}{\sqrt{y_1^2+y_2^2}}$, with static parameters as described in the introduction to this section. **SGD** was used to generate this data, with chunks of points as described earlier.

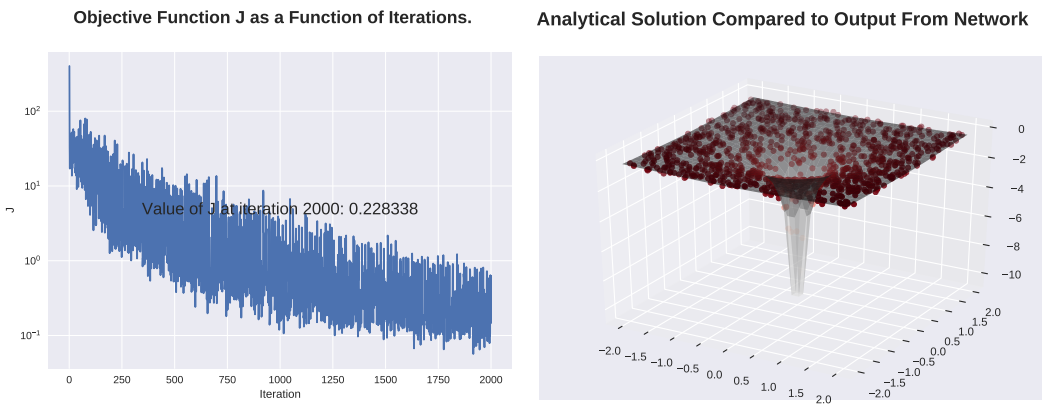
K	d	h	J
30	4	0.2	0.08201572476933472
23	4	0.2	0.10809256686691317
30	4	0.3	0.14068618388611653
15	4	0.2	0.2302819009741516
15	3	0.3	0.2490357105315411
30	4	0.1	0.24977744806088803

Table 16: Top 6 combinations from **testing** with data from $F(y) = -\frac{1}{\sqrt{y_1^2+y_2^2}}$, with static parameters as described in the introduction to this section. All the points I were run through the network at once when generating this data.

K	d	h	J	Ratio
23	4	0.1	0.3625586191101451	0.166
20	4	0.1	1.2179603097113043	0.086
23	4	0.2	1.303982916779365	0.093
10	4	0.2	2.1202635539540613	0.084
20	3	0.2	2.1365853306803984	0.048
15	4	0.1	2.17527290242069	0.088

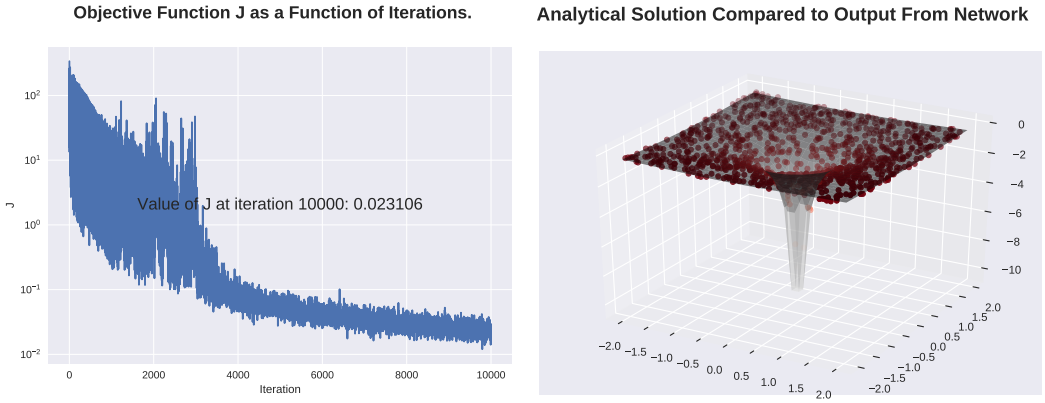
Table 17: Top 6 combinations from **testing** with data from $F(y) = -\frac{1}{\sqrt{y_1^2+y_2^2}}$, with static parameters as described in the introduction to this section. **SGD** was used in the training to generate this data, with chunks of points as described earlier.

K	d	h	J	Ratio
30	4	0.1	0.41265842753863485	0.118
30	4	0.2	0.5144479521090917	0.123
20	4	0.1	0.794004860459447	0.135
15	4	0.1	0.8524836740223009	0.098
30	4	0.05	1.0659272221439833	0.089
30	3	0.1	1.2473744532299504	0.124



(a) Objective function (b) Comparison

Figure 6: Figures produced by training and testing a neural network with the upper combination of values from table 17. Figure (a) shows the objective function with respect to the iterations and figure (b) shows the exact solution compared to the scatter plot produced by the network. This qualitatively shows that the approximation is alright, but figure 7 shows that more iterations is especially preferable in this two-dimensional case.



(a) Objective function (b) Comparison

Figure 7: Figures produced by training and testing a neural network with the upper combination of values from table 17, with 10000 iterations instead of 2000. Figure (a) shows the objective function as a function of iterations and the figure (b) shows the exact solution together with the scatter plot produced by the network. This qualitatively shows that the approximation is better with more iterations. The ratio of correctly classified points in this realization is 0.19, which is better than the uppermost value in table 17.

3.5 Discussion and Conclusion

Is it possible to infer which parameters are the best and hence which parameters should be chosen for the rest of the cases in the project? The not-so definite answer to this question

is perhaps. It is important to note that these functions are most useful for simply testing whether or not the implementation of the network is correct. Also, they indicate that the neural network is useful for the main task at hand, which is working with Hamiltonian systems, since they give good qualitative approximations, at least of these simple functions. The ratios of the one-dimensional cases look promising with the tolerance 0.005 that was used, while the tolerance is too strict for the two-dimensional case to give high ratios (running the tests with tolerance of 0.05 gives much better ratios for test function 3 and 4).

From the data one can gather that a fair part of the combinations of parameters tested can be disregarded in the following, based on their performance on the test functions. These combinations are the ones not part of the tables presented, since they did not make it to top six in their respective simulations.

An observation is that a higher dimension of the function's input requires more iterations for the objective function to approach zero and thus more iterations to achieve a good approximation of the function. For example, from figure 4 we see that the second test function (with 1-dimensional input) only requires 2000 iterations to achieve an objective function of magnitude 10^{-3}, while in figure 7 we see that the fourth test function requires about 10 000 iterations to achieve an objective function of magnitude 10^{-2}. This is expected, as a higher dimensional input implies more degrees of freedom when determining an approximation to the function output. As a side note, the run time naturally increased when increasing the iterations from 2000 to 10000. However, the change was not dramatic, and the tests that were run could be run with a higher amount of iterations than 2000 without the run time becoming unmanageable. Based on the observations in the two-dimensional cases, it is expected that the amount of iterations will probably need to be raised even higher when the dimension increases, as is the case with the supplied data from unknown Hamiltonian.

Conclusively, it is difficult to choose a definite optimal combination of parameters for the model, that could be counted on in the forthcoming. However, with an emphasis on the data from the two last test functions, since the majority of the problems to come are in higher dimensions than one, the combination of $K = 30$, combined with $d = 4$ or $d = 3$ and $h = 0.1$ or $h = 0.05$ stands out from the data. Therefore, they have been tested for the problems to come. It shows that they give alright results also in the forthcoming. This does not mean that we will exclusively use these combinations of parameters. In cases where they give sub-optimal results, other combinations shall be used as well.

3.6 Improvements

The testing process chosen can be vastly enhanced and the testing can be done more extensively, if time and resources permits. One could test a much wider range of parameters, to increase the certainty about which parameters are optimal. One could have run tests with more points at once and with more iterations. This gives a higher run time, which is why we chose to be restrictive, since our computing power is limited. The exact choice of 2000 iterations and 1000 points were a bit random, but the data generated can still demonstrate some general patterns in what combinations work the best, which makes the analysis worthwhile.

Finally, the data displayed is from one realization of training and testing the neural networks with the given parameter combinations. An average, mean or a statistical approach to the investigation may have given somewhat different results, but this is more demanding and is not something we have prioritized in the investigation. This is something one should have in mind when forming conclusions from the tests.

4 Given Hamiltonians

This section presents results from working with the supplied examples of separable Hamiltonian problems in mechanics. The numerical methods symplectic Euler (henceforth Euler) and Størmer-Verlet (henceforth SV) have been implemented, both with the use of the exact gradients of each of the problems and with the approximate gradients calculated from the trained networks. We introduce the notation

$$\widetilde{H}(q,p) = \widetilde{T}(p) + \widetilde{V}(q) \tag{2}$$

to represent the approximate Hamiltonian given by the two trained networks $\widetilde{T}(p)$ and $\widetilde{V}(q)$. Similarly, $H(p,q) = T(p) + V(q)$ is the exact Hamiltonian. q_0 and p_0 will denote the initial coordinates and momenta, respectively. For each of the problems the positional (q) coordinates and the coordinates of the momentum (p) will be displayed in figures. Moreover, the Hamiltonian will be plotted for each of the numerical methods, both with exact gradient and approximate gradient. A discussion will ensue from these figures, where we shall see to what extent the numerical solution preserves the Hamiltonian along trajectories in each case.

4.1 Derivation of the Gradient Formula

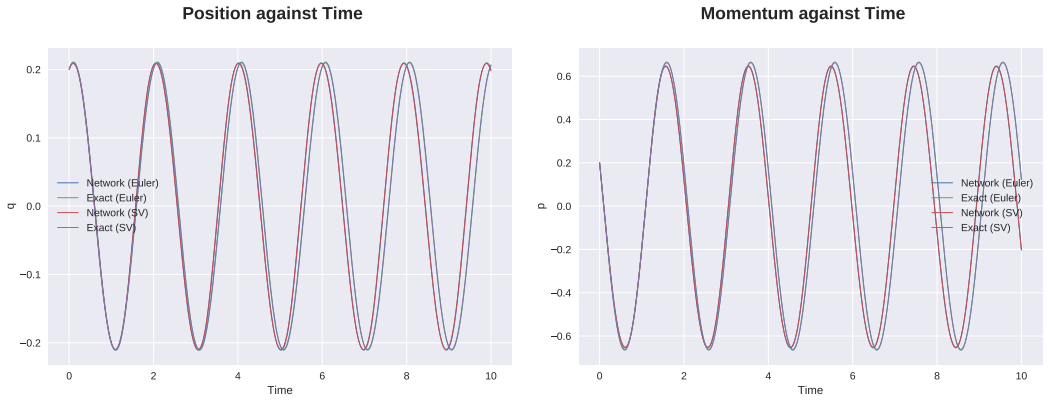
In order to implement numerical methods utilizing $\nabla\widetilde{H}$ calculate trajectories in time, one needs to derive a procedure for computing the gradient from the network. From the notation of the project description we need to find $\nabla_y \widetilde{F}(y)$. Using the formulas presented in the Project 2 Supplement, we find that $\nabla G(Z^{(K)}) = w \cdot \eta'(w^T Z^{(K)} + \mu)$ and $D\Phi_{k-1}(Z^{(k-1)})^T A = A + W_{k-1}^T(h\sigma'(W_{k-1}Z^{(k-1)} + b_{k-1}) \odot A)$. Thus, a procedure for computing the gradient is

$A \leftarrow w \cdot \eta'(w^T Z^{(K)} + \mu)$
for $k = K, K-1, \ldots, 1$ **do**
  $A \leftarrow A + W_{k-1}^T(h\sigma'(W_{k-1}Z^{(k-1)} + b_{k-1}) \odot A)$
end for

where w is a $d \times 1$ vector and in the first line we compute the standard inner product between w and $\eta'(\cdot)$. This yields a $d \times I$ matrix. When the for-loop is done, the j^{th} column of A represents the gradient corresponding to the j^{th} input point, y_j. The gradient to the input point y_j has d elements. We therefore use the d_0 first elements of the resulting gradient, analogous to how we embed the input data in d dimensions.

4.2 Nonlinear Pendulum Problem

Results from the nonlinear pendulum problem are shown in figures 8 and 9. The networks were trained with the parameters $K = 30$, $d = 4$, $h = 0.1$, $I = 1000$ and 5000 iterations. In the numerical methods we used 1000 time steps between time 0 and 10, with initial values $p_0 = q_0 = 0.2$.



(a) q(t) (b) p(t)

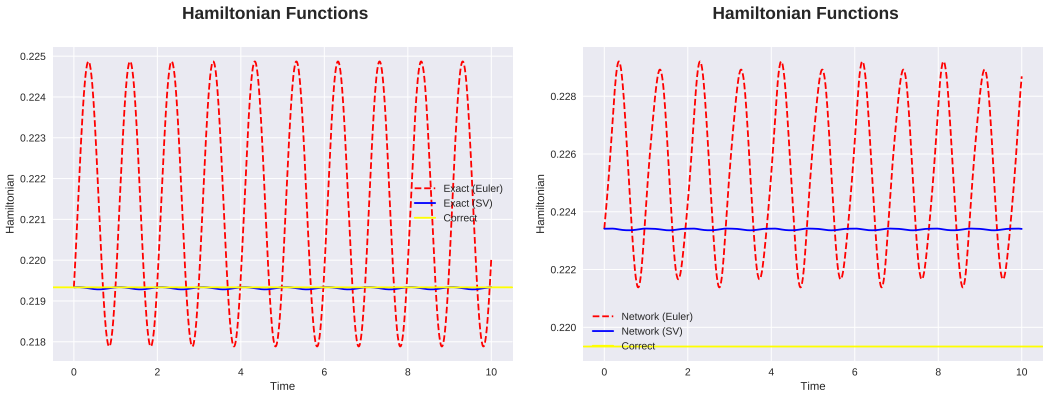
r

Figure 8: Figure (a) shows position plotted against time. Figure (b) shows momentum plotted against time. Both the positions and the momenta are calculated using both Euler and SV, both with exact gradient and with approximate gradient. The label "Network" refers to the approximate gradient, while the label "Exact" refers to the exact gradient.

From figure 8 we see that the exact methods produce nearly the same trajectory for this setup of the nonlinear pendulum. We can also see that the trajectories produced with the network-gradients are almost indistinguishable and very close to the exact trajectories.

In figure 9 we have plotted the Hamiltonian along the trajectories produced by the exact methods (figure 9a) and the network methods (figure 9b). It is apparent that the Hamiltonian calculated from the two numerical methods (SV and Euler) are very similar regardless of which gradient is used (exact or from network). The most noticeable difference between the figures is that the Hamiltonians produced from the network looks "shifted" away from the correct Hamiltonian. This is probably due to the error in the network, while the sinusoidal oscillation is most likely caused by the error in the numerical methods.

We can conclude that the numerical methods using the network for calculating the gradients give a reasonable approximation to the exact numerical methods. Hence, the gradient is approximated fairly well by the network in this problem. We also note that SV preserves the Hamiltonian to a much higher degree than Euler, which exhibits a much greater amplitude in the oscillations around the correct Hamiltonian.



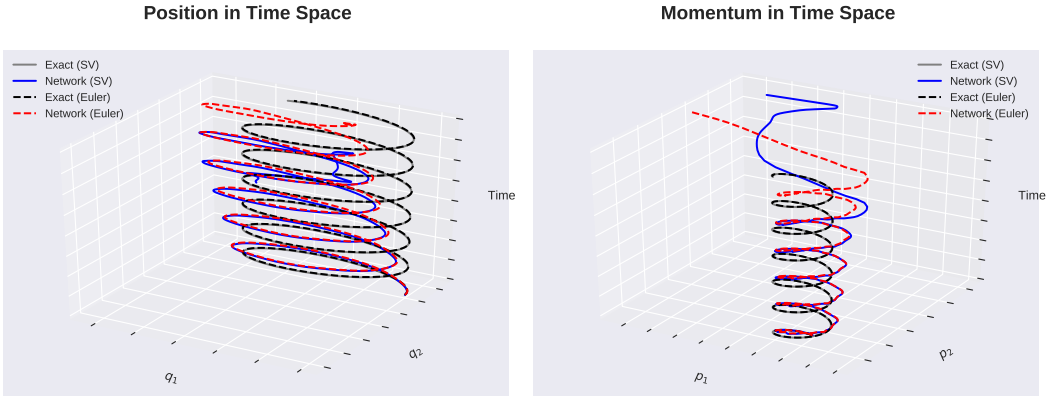
(a) Hamiltonian - exact gradient

(b) Hamiltonian - approx. gradient

Figure 9: Both figures show the correct Hamiltonian of the system, $H(p_0, q_0)$ (yellow). Figure (a) shows $H(p, q)$ along the trajectories calculated with the exact gradient, ∇H, with SV (blue) and with Euler (red dashed). Similarly, figure (b) shows $\widetilde{H}(p, q)$ along the trajectories calculated with $\nabla \widetilde{H}$, with SV (blue) and with Euler (red dashed).

4.3 Kepler's Two-Body Problem

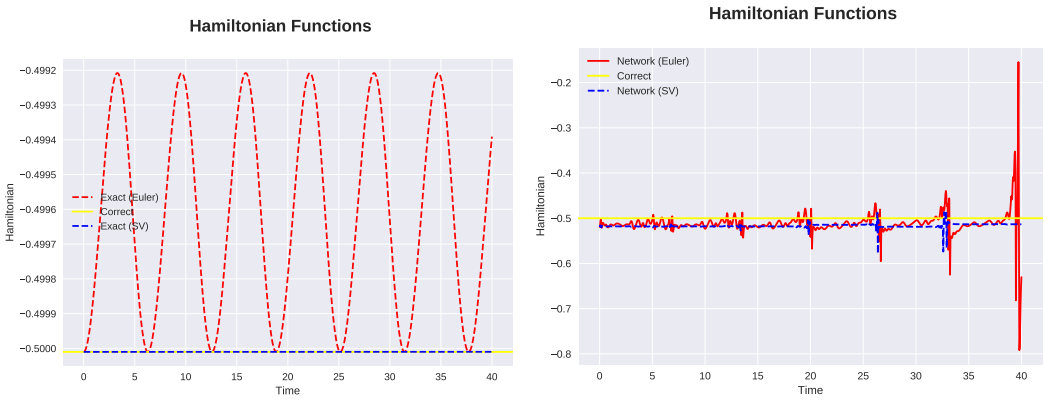
Results from the Kepler two-body problem are shown in figure 10 and 11. The networks were trained with the parameters $K = 30$, $d = 4$, $h = 0.1$, $I = 1000$ and 5000 iterations. In the numerical methods we used 1000 time steps between time 0 and 40, with initial values $q_0 = [1 - c, 0]^T$ and $p_0 = [0, \sqrt{(1 + c)/(1 - c)}]^T$, where $c = 0.00001$.



(a) q(t)

(b) p(t)

Figure 10: Figure (a) shows position plotted against time. Figure (b) shows momentum plotted against time. Both the positions and the momenta are calculated using both Euler and SV, both with exact gradient and with approximate gradient. The label "Network" refers to the approximate gradient, while the label "Exact" refers to the exact gradient.



(a) Hamiltonian - exact gradient

(b) Hamiltonian - approx. gradient

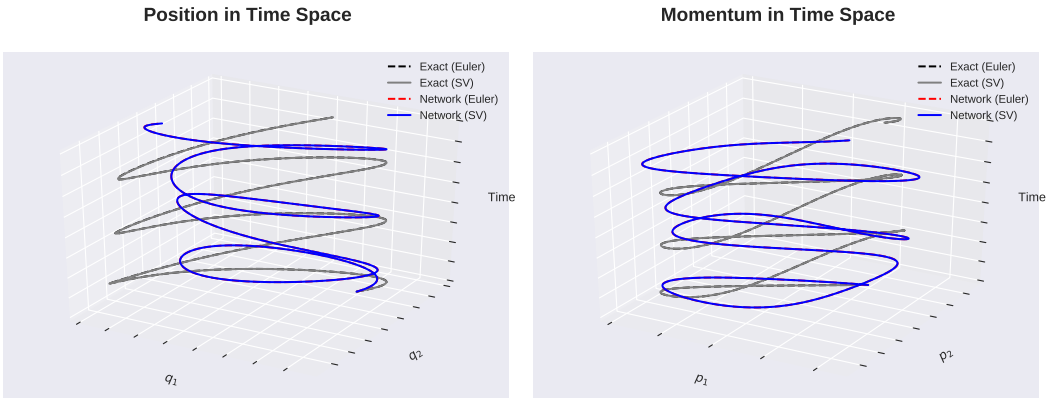
Figure 11: Both figures show the correct Hamiltonian of the system, $H(p_0, q_0)$ (yellow). Figure (a) shows $H(p, q)$ along the trajectories calculated with the exact gradient, ∇H, with SV (blue dashed) and with Euler (red dashed). Similarly, figure (b) shows $\widetilde{H}(p, q)$ along the trajectories calculated with $\nabla \widetilde{H}$, with SV (blue) and with Euler (red dashed).

From figure 10 we see that the network methods and exact method produce similar trajectories in shape, but we also notice that the network trajectories seem to diverge at the end of the time interval. We also see this when we plot $\widetilde{H}$ in figure 11b. For both SV and Euler, the amplitude of $\widetilde{H}$ increases with time, which is connected to the fact that the network solution "wanders away" in figure 10. This issue could perhaps be mitigated by scaling the input and output of the network.

Once again it is apparent from figure 10 that Euler is more unstable than SV and leads to a worse approximation to the correct, analytical solution, compared to SV.

4.4 Henon-Heiles Problem

Results from the Henon-Heiles problem are shown in figure 12 and figure 13. The networks were trained with the parameters $K = 30$, $d = 4$, $h = 0.1$, $I = 1000$ and 5000 iterations. In the numerical methods we used 5000 time steps between time 0 and 20, with initial values $p_0 = q_0 = [0.2, 0.2]^T$.

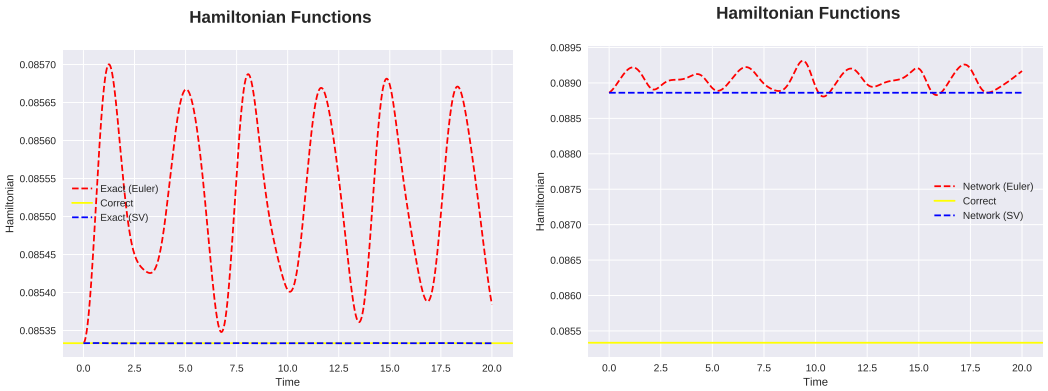


(a) q(t) (b) p(t)

Figure 12: Figure (a) shows position plotted against time. Figure (b) shows momentum plotted against time. Both the positions and the momenta are calculated using both Euler and SV, both with exact gradient and with approximate gradient. The label "Network" refers to the approximate gradient, while the label "Exact" refers to the exact gradient.

It is hard to read how closely related the graphs in figure 12 actually are, but when running the code that produces the plots one can rotate it and get a better look. It then becomes apparent that the two graphs in each of the subplots are somewhat closely related with regard to the general shape. As before, this indicates that $\nabla \widetilde{H}$ can be used as a reasonable approximation to the exact gradient. Comparing to the nonlinear pendulum, the results indicate yet again that the approximation is better in one dimension compared to in two dimensions.

When it comes to the difference between Euler and SV, we again see in figure 13 that Euler is worse at preserving the Hamiltonian along the trajectories than SV.



(a) Hamiltonian - exact gradient

(b) Hamiltonian - approx. gradient

Figure 13: Both figures show the correct Hamiltonian of the system, $H(p_0, q_0)$ (yellow). Figure (a) shows $H(p, q)$ along the trajectories calculated with the exact gradient, ∇H, with SV (blue dashed) and with Euler (red dashed). Similarly, figure (b) shows $\widetilde{H}(p, q)$ along the trajectories calculated with $\nabla \widetilde{H}$, with SV (blue dashed) and with Euler (red dashed).

4.5 Further Discussion on Numerical Methods

Across all the examples above, we have seen that SV preserves the Hamiltonian along trajectories to a higher degree than Euler. This is as expected, based on the difference in order between the two methods. We also note that the preservation of the Hamiltonian is sensitive to the number of time steps on the interval in question. An example of this is shown in figure 14, where we simulate the same trajectory as in figure 9a, but with 100 time steps in stead of 1000. Especially SV oscillates with a greater amplitude. In the next section, however, we will look at trajectories with 2047 time steps on the interval $t \in (0, 10)$, so one needs probably not to be concerned about having enough time steps in that case. Based on the above discussion, we will be using SV when calculating trajectories in the proceeding section, since it gives better approximations compared to Euler.

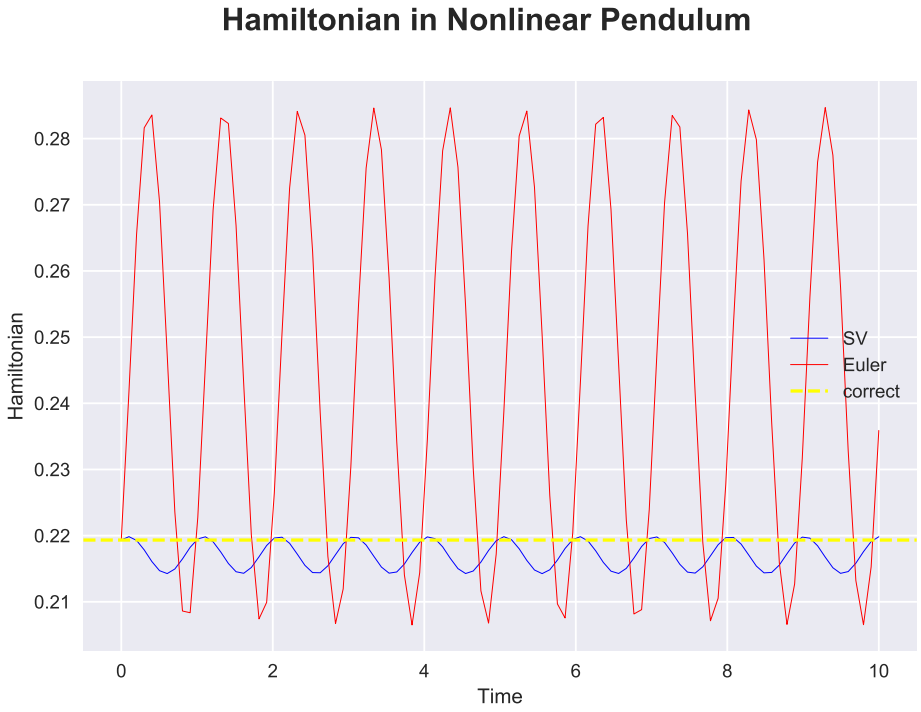


Figure 14: The Hamiltonian in the nonlinear pendulum problem along trajectories calculated with SV and Euler. The inital conditions are the same as in section 4.2, but there are 100 time steps instead of 1000.

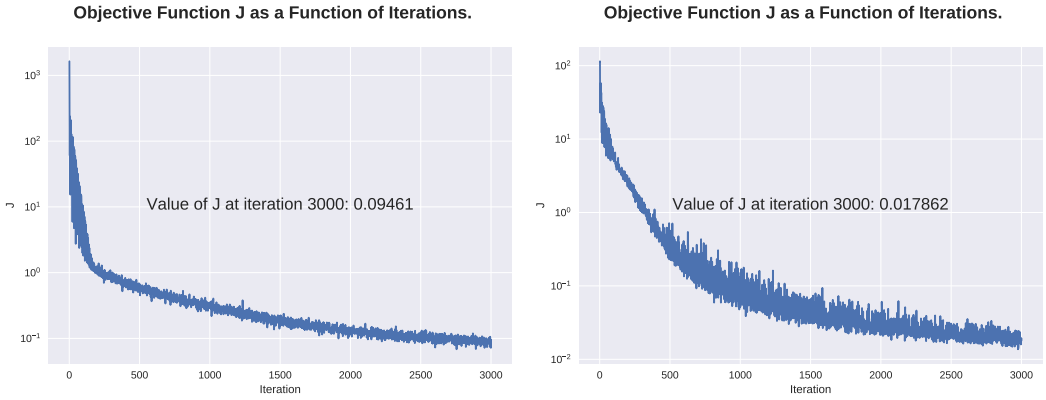
5 Data - Unknown Hamiltonian

In this section, we will try to approximate the trajectories in a Hamiltonian system where the Hamiltonian itself is unknown. To do this, we have a large set of trajectories computed from this unknown source, that we will train our networks with. Then we use SV, as in section 4, to calculate the trajectories. In the following, two examples of the calculated trajectory are presented.

5.1 Example 1

In this example, we try to predict the trajectory of batch number 39. The parameters used were $K = 30$, $h = 0.1$, $d = 4$, $\bar{I} = I/256 = 480$ points with SGD and 3000 iterations. The training was done with the 30 first batches of the supplied data. One network is made for the kinetic energy, $\widetilde{T}(p)$, and another network is made for the potential energy, $\widetilde{V}(q)$. This is necessary when calculating the gradients, since one depends on the momenta and the other depends on the positions. Figure 15 displays the convergence plots from training the two networks. We note that the run time is very large when running the algorithm without SGD in this case, since the provided data contains many points. It is significantly improved when using SGD, but it still takes longer than experienced when training and testing the

suggested functions in part 3 of the report.



(a) Network $\widetilde{V}$ (b) Network $\widetilde{T}$

Figure 15: Convergence plots of the objective function when training neural networks for V and T based on the given data.

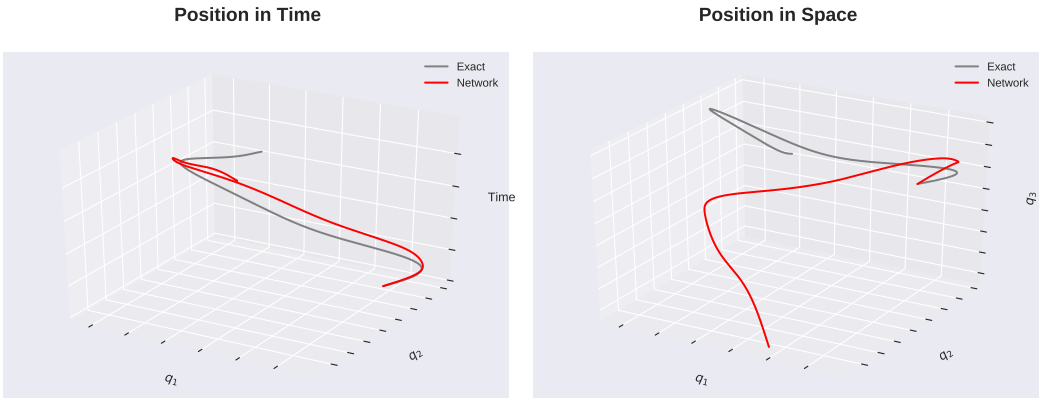
Comparing the convergence plots in figure 15 with the convergence plots for the two-dimensional cases in section 4 (figure 5, 6 and 7) the objective functions converge well.

The ratios of correctly classified points after training, within defined tolerance tol $= 0.05$, analogous to when testing the suggested test functions in part 4, are given by $V_{\text{ratio}} = 0.4083$ and $T_{\text{ratio}} = 0.0875$. This ratio indicates that the classification of the points is weak.

Figure 16 shows the positional values calculated from the network using SV, after testing the trained networks presented earlier on batch 39, as well as the correct trajectory from the supplied data. Similarly, figure 17 shows the values of the momentum calculated from the network using SV, after testing, as well as the correct trajectory of the momentum from the supplied data.

Figure 16a shows that the two first positional coordinates fairly accurately approximates the correct coordinates. Despite this, figure 16b indicates that all three coordinates together do not approximate the correct trajectory in space very closely.

Running the simulations with a bigger chunk, e.g. $I/128 = 960$ points gives slightly increased ratios, but the run time also increases. In the limit, when all the points are run through at once, i.e. when the optimization is done without SGD, the run time is very large. Therefore, SGD was a natural choice here, since the supplied data has many points.

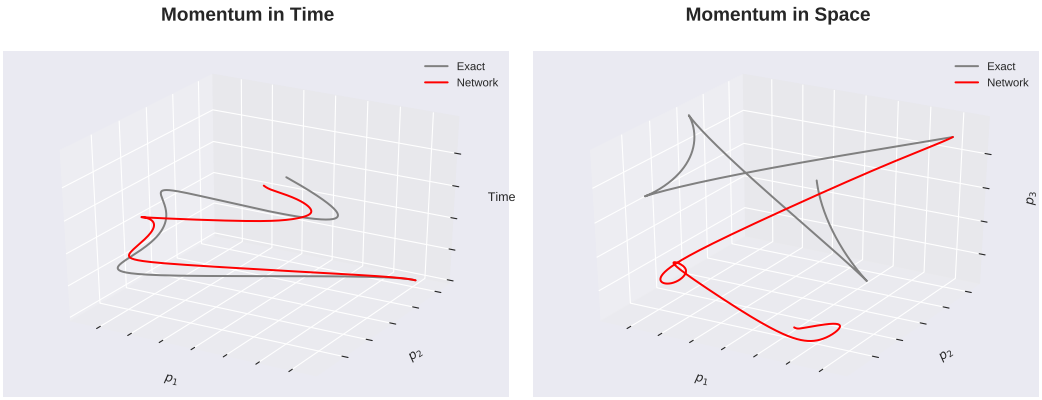


(a) q_1, q_2 and time

(b) q_1, q_2 and q_3

Figure 16: Comparison between the positions of the correct solution and the solution produced based on the network. Figure (a) shows the two first positional coordinates plotted against time for the exact data (grey) and calculated approximation from network (red). Figure (b) shows all three positional coordinates of the same quantities with the same color labelling.

Momentum values tested on the same batch are shown in figure 17. Similar observations as for the positional coordinates can be made about the momentum.



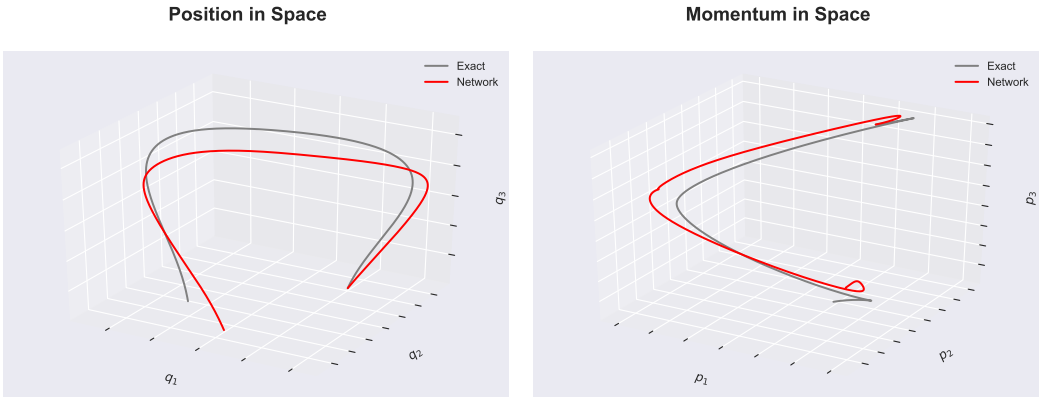
(a) p_1, p_2 and time

(b) p_1, p_2 and p_3

Figure 17: Comparison between the momentum of the correct solution and the solution produced based on the network. Figure (a) shows the two first coordinates of the momentum plotted against time for the exact data (grey) and calculated approximation from network (red). Figure (b) shows all three coordinates of the momentum of the same quantities with the same color labelling.

5.2 Example 2

In this example, we are trying to calculate the trajectory of batch number 45. To do this, the networks $\widetilde{T}(p)$ and $\widetilde{V}(q)$ are trained on the 41 first batches, with parameters $K = 30$, $h = 0.1$, $\bar{I} = I/256$ and 3000 iterations.



(a) q_1, q_2 and q_3

(b) p_1, p_2 and p_3

Figure 18: Figure (a) shows the positional coordinates of the trajectory for the given data and for the network. Figure (b) shows the momentum-coordinates for the given data and for the network

In figure 18, the exact positions and momenta for the trajectory is plotted with the approximations computed with $\widetilde{H}$. To get a better impression of how the exact data compares to the network, the individual positional coordinates are plotted against time for the exact- and network-solution in figure 19.

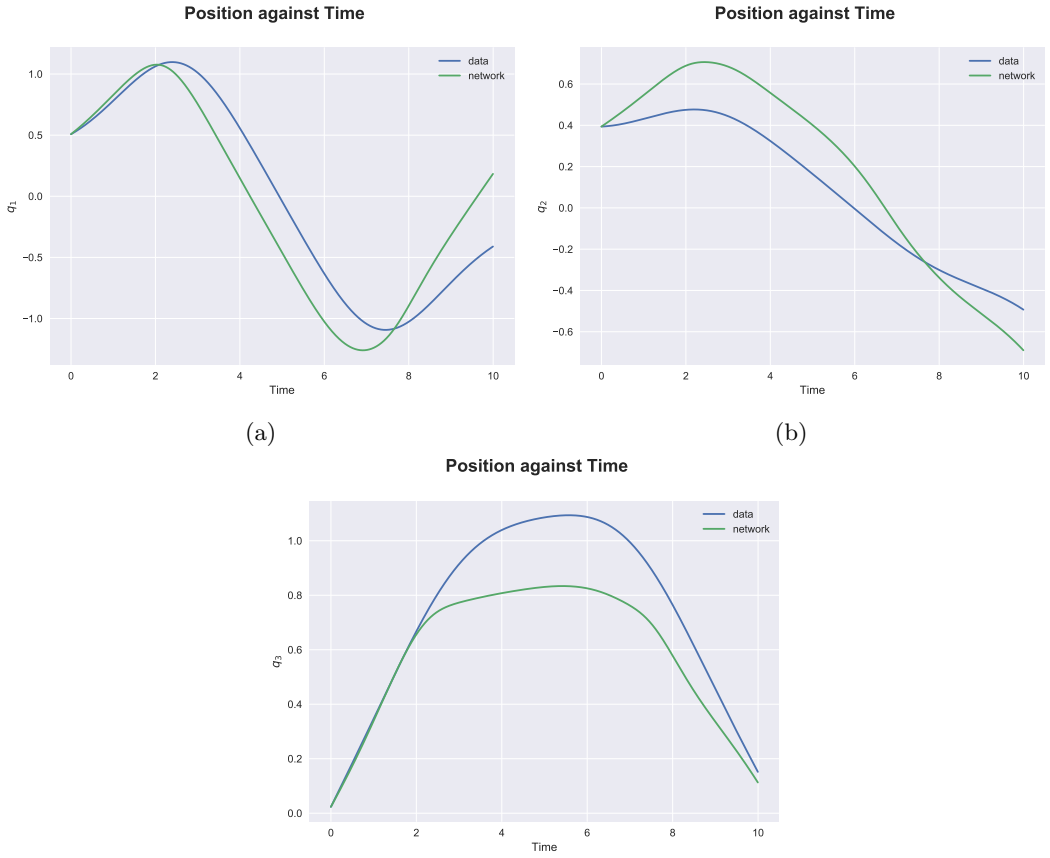


Figure 19: (a) The first positional coordinate, q_1, plotted against time. (b) The second positional coordinate, q_2, plotted against time. (c) The third positional coordinate, q_3, plotted against time.

In figure 20, $\widetilde{H}$ is plotted along the trajectory calculated with the network. We see that it is practically constant, and fairly close to the Hamiltonian of the exact trajectory.

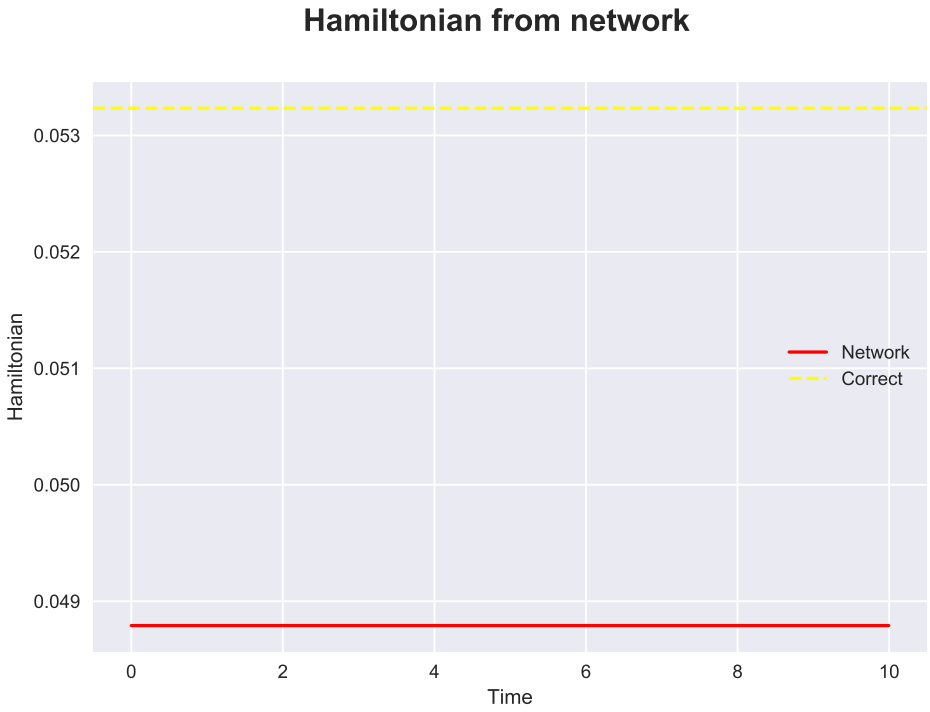


Figure 20: The network Hamiltonian, $\widetilde{H}$, along the trajectory calculated with the network (red). The yellow dashed line is the correct Hamiltonian for the provided trajectory.

5.3 Discussion of results

Although the testing presented here is limited, we see that this method of using neural networks to calculate trajectories with numerical methods gives some promising results. Especially in Example 2, we see that the approximate trajectory is close to the provided one.

If we had more time on our hands, we could try to achieve more accurate results by looking at the discrepancy between the network trajectory and the provided trajectory, and try to minimize this by tweaking the parameters in the network, θ. One could, in a similar manner, perhaps tweak the parameters so that $\widetilde{H}$ along the network trajectory comes closer to H for the exact trajectory.

6 Code Architecture

First of all we chose to hand in the code in separate Python-files instead of in a Jupyter Notebook, because there is a lot of code that has very different use cases. This means that there are a lot of different files, which might seem unorganized. However, we have done our best to label the files, functions and classes in ways that are easy to understand, and we have tried to organize the code in a logical way. Moreover, each file has a description in the beginning, in case the file names are not descriptive enough. Furthermore, each function and class have docstrings, to raise the abstraction level. In this way the reader of the code

hopefully does not need to focus on the implementation details, but should instead be able to use the interface easily without focusing on how the interface itself is implemented. In the case that the reader wants to understand all implementation details, the code has been written as cleanly as possible where it seemed reasonable.

Besides the strictly necessary code for working with the neural network, some of the code we wrote is used for running tests on the model and processing the resulting data as easily and efficiently as possible. In section 3, we ran a significant amount of simulations for different combinations of parameters and chose to display only some examples that were indicative of the behaviour and performance of the model. Most of the simulations were run on the "calculation computer" Markov in the math department and we wrote code to make sorted csv-files based on the value of the objective function in the last iteration. Furthermore, the csv-files were automatically inserted as tables into LaTeX. This "extra" work really has nothing to do with the task at hand, as in it was not needed to complete the task, but still made it very easy for us to gain access to vast amounts of data and to display some of it in this report. Therefore, we were able to test the model thoroughly.

6.1 Overview of Files

A quick overview of the submitted files used in each section is given in the following. Keep in mind that some of the files are used in many sections throughout the report, but we have tried to organize them here as best as possible to make it easier for you to keep track of each file.

The file `network_algo.py` contains the class for the neural network and the different algorithms for training the network. This is the corner stone of the software. Most of the other files are composed of code to analyze the neural network. The file `test_model` is another file that is used throughout, to run tests on the network. The other files are listed below.

6.1.1 Model Choices

- `vanilla_vs_adams.py`
- `scaling_test.py`

6.1.2 Results and Discussion Regarding Suggested Functions

- `systematic_tests4_test_functions.py`
- `run_systematic_tests.py`
- `data_processing.py`
- `test_functions.py`
- `run_tests.sh`

6.1.3 Given Hamiltonians

- `hamiltonian_problems_ex.py`
- `numerical_methods.py`
- `test_numerical_methods.py`

6.1.4 Data - Unknown Hamiltonian

- `test_given_data.py`
- `import_data.py` (supplied code)